Relational algebra: a Kleene algebra central to the mathematics of program construction

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

II Jornadas Luso-Galegas de Álgebra e Computação
Braga, 10-10-2008

On maths and computing

Interaction between maths and computing:

- computers helping maths: theorem proving, computational maths etc
- maths helping computing: many examples, among which the algebra of programming (**AoP**)

While the former are widely acknowledged, among the latter **AoP** is known only to the initiated.

- This talk aims at framing **AoP** in its proper algebraic context while showing its relevance to program construction.

It all starts from semirings of computations [3]...

On maths and computing

Interaction between maths and computing:

- computers helping maths: theorem proving, computational maths etc
- maths helping computing: many examples, among which the algebra of programming (**AoP**)

While the former are widely acknowledged, among the latter **AoP** is known only to the initiated.

- This talk aims at framing **AoP** in its proper algebraic context while showing its relevance to program construction.

It all starts from semirings of computations [3]...

Semirings of computations

Abstract notion of a computation:

Semiring $(S, +, \cdot, 0, 1)$ *inhabited by computations (eg. instructions, statements) where*

- $x \cdot y$ *(usually abbreviated to* xy*) captures* **sequencing**
- $x + y$ *captures* **choice** *(alternation)*
- 0 *means* **death**
- 1 *means* **skip** *(do nothing)*

Technically:

- $(S, \cdot, 1)$ is a monoid
- $(S, +, 0)$ is a Abelian monoid
- $(\cdot)$ distributes over $(+)$
- 0 annihilates $(\cdot)$

Idempotency

- If $x + x = x$ holds for all x, then

$$x \leq y \stackrel{\mathrm{def}}{=} x + y = y \tag{1}$$

  is a partial order.

- Clearly, $0 \leq x$ for all x and $(+)$ is the *lub* with respect to $\leq$:

$$x + y \leq z \iff x \leq z \wedge y \leq z \tag{2}$$

  NB: $z := x + y$ in (2) means $x + y$ is upper bound; $\Leftarrow$ means it is the **least** upper bound (*lub*).

Kleene algebras

A Kleene algebra [5] adds to semiring $(S, +, \cdot, 0, 1)$ the *Kleene star* operator $(^*)$ such that

$$y + x(x^* y) \leq x^* y \quad (3)$$

$$y + (y x^*) x \leq y x^* \quad (4)$$

and

$$y + xz \leq z \Rightarrow x^* y \leq z \quad (5)$$

$$y + zx \leq z \Rightarrow y x^* \leq z \quad (6)$$

These basically establish $x^* y$ and $y x^*$ as prefix points of (monotonic) functions $(y + x \cdot \_)$ and $(y + \_ \cdot x)$, respectively.

KATs (tests and domains)

KAT = Kleene algebra with tests

- every p below 1 ($p \leq 1$) is a **test** and such that, for every such p there is $\neg p$ (the *complement* of p) such that

$$p + \neg p = 1$$
$$p \cdot \neg p = 0 = \neg p \cdot p$$

- Recent addition to semirings (inc. KATs) of a *domain* operator $d(x)$ capturing "enabledness" and satisfying axioms

$$\begin{array}{rcl} d(x) & \leq & 1 \\ d(0) & = & 0 \\ d(x+y) & = & d(x) + d(y) \\ d(xy) & = & d(x\ d(y)) \\ x & \leq & d(x)x \end{array}$$

Binary relations

The algebra of **binary relations** is a well known KAT:

KAT	Binary relations	Description
$x \cdot y$	$R \cdot S$	composition
$x + y$	$R \cup S$	union
0	$\bot$	empty relation
1	id	identity relation
$x \leq y$	$R \subseteq S$	inclusion
$p, \neg p$	$R \subseteq id$, $\neg R = id - R$	coreflexive relations
$d(x)$	$\delta\, R$	domain of R

Moreover, they form a complete, distributive lattice once *glb*s

$$X \;\subseteq\; R \cap S \quad\Leftrightarrow\quad (X \subseteq R) \wedge (X \subseteq S) \tag{7}$$

and supremum $\top$ are added.

How useful are binary relations?

- Not much if regarded merely as "sets of pairs"
- Very useful indeed — as a device for the algebraization of logic — if regarded as "**arrows**" ie. morphisms of a particular allegory [4]
- Arrows bring about a **type discipline** which leads to good things such as parametric **polymorphism**, etc etc

Relations as morphisms

Binary relations are typed:

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation from A (source) to B (target).

A, B are types. Writing $B \xleftarrow{R} A$ means the same as $A \xrightarrow{R} B$.

Infix notation

The usual infix notation used in natural language — eg.

$$\texttt{John}\ \mathit{IsFatherOf}\ \texttt{Mary}$$

— and in maths — eg.

$$0 \leq \pi$$

— extends to arbitrary $B \xleftarrow{R} A$: we write

$$b\ R\ a$$

to denote that $(b, a) \in R$.

Relations as morphisms

Binary relations are typed:

Arrow notation

Arrow $A \xrightarrow{R} B$ denotes a binary relation from A (source) to B (target).

A, B are types. Writing $B \xleftarrow{R} A$ means the same as $A \xrightarrow{R} B$.

Infix notation

The usual infix notation used in natural language — eg.

$$\texttt{John}\ \mathit{IsFatherOf}\ \texttt{Mary}$$

— and in maths — eg.

$$0 \leq \pi$$

— extends to arbitrary $B \xleftarrow{R} A$: we write

$$b\ R\ a$$

to denote that $(b, a) \in R$.

Functions are relations

- Lowercase letters (or identifiers starting by one such letter) will denote special relations known as **functions**, eg. f, g, suc, etc.
- We regard **function** $f : A \longrightarrow B$ as the binary **relation** which relates b to a iff $b = f\ a$. So,

$$b\ f\ a \text{ literally means } b = f\ a$$

- Therefore, we generalize

$$\boxed{\begin{array}{c} B \xleftarrow{f} A \\ b = f\ a \end{array}} \quad \text{to} \quad \boxed{\begin{array}{c} B \xleftarrow{R} A \\ b\ R\ a \end{array}}$$

- So, **function** id is the equality (equivalence) **relation**:

$$b\ id\ a \text{ means the same as } b = a$$

Composition

Function composition

$$\begin{tikzcd} B & A \arrow[l, "f"'] & C \arrow[l, "g"'] \arrow[ll, bend left, "f \cdot g"] \end{tikzcd} \tag{8}$$

$$b = f(g\ c)$$

extends to $R \cdot S$ in the obvious way:

$$b(R \cdot S)c \quad \Leftrightarrow \quad \langle \exists\ a\ ::\ b\ R\ a \wedge a\ S\ c \rangle \tag{9}$$

Note how this rule *removes* quantifier $\exists$ when applied from right to left.

Converses

Every relation $B \xleftarrow{R} A$ has a **converse** $B \xrightarrow{R^\circ} A$ which is such that, for all a, b,

$$a(R^\circ)b \quad \Leftrightarrow \quad b\ R\ a \tag{10}$$

Note that converse commutes with composition

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{11}$$

and cancels itself

$$(R^\circ)^\circ = R \tag{12}$$

Function converses

Function converses f°, g° etc. always exist (as **relations**) and enjoy the following (very useful) property:

$$(f\ b)R(g\ a) \quad \Leftrightarrow \quad b(f^\circ \cdot R \cdot g)a \tag{13}$$

cf. diagram:

$$\begin{tikzcd}
C & D \arrow[l, "R"'] \\
B \arrow[u, "f"] & A \arrow[l, "f^\circ \cdot R \cdot g"] \arrow[u, "g"']
\end{tikzcd}$$

Why *id* (really) matters

Terminology:

- Say R *is* *reflexive* iff $id \subseteq R$
  pointwise: $\langle \forall\ a\ ::\ a\ R\ a \rangle$
- Say R *is* *coreflexive* iff $R \subseteq id$
  pointwise: $\langle \forall\ b, a\ :\ b\ R\ a :\ b = a \rangle$

Define, for $B \xleftarrow{R} A$:

Kernel of R	**Image** of R
$A \xleftarrow{\ker R} A$ $\ker R \triangleq R^\circ \cdot R$	$B \xleftarrow{\mathrm{img}\, R} B$ $\mathrm{img}\, R \triangleq R \cdot R^\circ$

Example

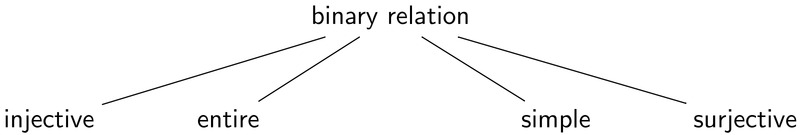
Kernels of functions:

$$
\begin{array}{ll}
 & a'(\ker f)a \\
\Leftrightarrow & \quad \{ \text{ substitution } \} \\
 & a'(f^{\circ} \cdot f)a \\
\Leftrightarrow & \quad \{ \text{ PF-transform rule (13) } \} \\
 & (f\ a') = (f\ a)
\end{array}
$$

In words: $a'(\ker f)a$ means a' *and* a *"have the same* f*-image"*

Binary relation taxonomy

Topmost criteria:



Definitions:

	Reflexive	*Coreflexive*
$\ker R$	entire R	injective R
$\mathrm{img}\, R$	surjective R	simple R

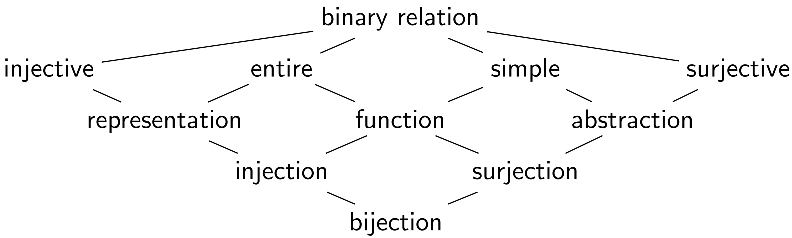
(14)

Facts:

$$\ker (R^\circ) = \mathrm{img}\, R \tag{15}$$
$$\mathrm{img}\, (R^\circ) = \ker R \tag{16}$$

Binary relation taxonomy

The whole picture:



(17)

Clearly:

- converse of *injective* is *simple* (and vice-versa)
- converse of *entire* is *surjective* (and vice-versa)
- smaller than injective (simple) is injective (simple)
- larger than entire (surjective) is entire (surjective)

Functions in one slide

A function f is a binary relation such that

Pointwise	**Pointfree**	
"Left" Uniqueness		
$b\ f\ a \wedge b'\ f\ a \ \Rightarrow\ b = b'$	$\mathrm{img}\, f \ \subseteq\ id$	(f is simple)
Leibniz principle		
$a = a' \ \Rightarrow\ f\ a = f\ a'$	$id \ \subseteq\ \ker f$	(f is entire)

which both together are equivalent to any of "al-gabr" rules

$$f \cdot R \subseteq S \ \Leftrightarrow\ R \subseteq f^{\circ} \cdot S \tag{18}$$

$$R \cdot f^{\circ} \subseteq S \ \Leftrightarrow\ R \subseteq S \cdot f \tag{19}$$

"Al-gabr" rules?

Recall *calculus of al-gabr and al-muqâbala* [1]:

al-gabr

$$x - z \leq y \quad \Leftrightarrow \quad x \leq y + z$$

al-hatt

$$x * z \leq y \quad \Leftrightarrow \quad x \leq y * z^{-1} \qquad (z > 0)$$

al-muqâbala

Ex:

$$4x^2 + 3 = 2x^2 + 2x + 6 \quad \Leftrightarrow \quad 2x^2 = 2x + 3$$

[1] Cf. *Kitâb al-muhtasar fi hisab al-gabr wa-almuqâbala* by Abû Al-Huwârizmî, the famous 9c Persian mathematician.

Example: function equality

Equating functions means comparing them in either way:

$$f = g \quad \Leftrightarrow \quad f \subseteq g \quad \Leftrightarrow \quad g \subseteq f \tag{20}$$

Calculation:

$$\begin{array}{ll}
 & f \subseteq g \\
\Leftrightarrow & \qquad \{ \text{ “al-gabr” (18) on } f \ \} \\
 & id \subseteq f^{\circ} \cdot g \\
\Leftrightarrow & \qquad \{ \text{ “al-gabr” (19) on } g \ \} \\
 & g^{\circ} \subseteq f^{\circ} \\
\Leftrightarrow & \qquad \{ \text{ converses } \} \\
 & g \subseteq f
\end{array}$$

A "Laplace transform analog" for logical quantification

The pointfree (PF) transform

ϕ	$PF\ \phi$
$\langle \exists\ a\ ::\ b\ R\ a \wedge a\ S\ c \rangle$	$b(R \cdot S)c$
$\langle \forall\ a, b\ ::\ b\ R\ a \Rightarrow b\ S\ a \rangle$	$R \subseteq S$
$\langle \forall\ a\ ::\ a\ R\ a \rangle$	$id \subseteq R$
$\langle \forall\ x\ ::\ x\ R\ b \Rightarrow x\ S\ a \rangle$	$b(R \setminus S)a$
$\langle \forall\ c\ ::\ b\ R\ c \Rightarrow a\ S\ c \rangle$	$a(S / R)b$
$b\ R\ a \wedge c\ S\ a$	$(b, c)\langle R, S \rangle a$
$b\ R\ a \wedge d\ S\ c$	$(b, d)(R \times S)(a, c)$
$b\ R\ a \wedge b\ S\ a$	$b\ (R \cap S)\ a$
$b\ R\ a \vee b\ S\ a$	$b\ (R \cup S)\ a$
$(f\ b)\ R\ (g\ a)$	$b(f^\circ \cdot R \cdot g)a$
TRUE	$b\ \top\ a$
FALSE	$b\ \bot\ a$

What do $\langle R, S \rangle$, $R \times S$ etc mean?

Forks for tupling

The **fork** ("split") combinator is essential for transforming predicates holding more than two quantified variables. From the definition,

$$(b, c)\langle R, S\rangle a \quad \Leftrightarrow \quad b\ R\ a \wedge c\ S\ a$$

which PF-transforms to

$$\langle R, S\rangle \quad = \quad \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \tag{21}$$

we infer diagram

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "R"] \arrow[u, "\langle R,S\rangle"] \arrow[ur, "S"'] &
\end{tikzcd}$$

and "al-gabr" rule (Galois connection)

$$\pi_1 \cdot X \subseteq R \ \wedge \ \pi_2 \cdot X \subseteq S \quad \Leftrightarrow \quad X \subseteq \langle R, S\rangle \tag{22}$$

Coproducts for *"if-then-else'ing"*

Define dual ("either") combinator as

$$[R\ ,S] \;=\; (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \quad \text{cf.} \quad \begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "R"'] & A + B \arrow[d, "{[R\ ,S]}"] & B \arrow[l, "i_2"'] \arrow[dl, "S"] \\ & C & \end{tikzcd}$$

From this and the *lub* rule (2) we infer another "al-gabr" rule (Galois connection)

$$[R\ ,S] \subseteq X \quad \Leftrightarrow \quad R \subseteq X \cdot i_1 \;\wedge\; S \subseteq X \cdot i_2 \tag{23}$$

In fact, the stronger universal property holds:

$$[R\ ,S] = X \quad \Leftrightarrow \quad R = X \cdot i_1 \;\wedge\; S = X \cdot i_2 \tag{24}$$

Multiplying and adding relations

From "fork" and "either" derive

$$R \times S \triangleq \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{25}$$

$$R + S = [i_1 \cdot R \;, i_2 \cdot S] \tag{26}$$

whose pointwise meaning is, as given earlier:

ϕ	$PF\ \phi$
$a\ R\ c \wedge b\ S\ c$	$(a, b)\langle R, S\rangle c$
$b\ R\ a \wedge d\ S\ c$	$(b, d)(R \times S)(a, c)$

Absorption properties:

$$\langle R \cdot X, S \cdot Y \rangle = (R \times S) \cdot \langle X, Y \rangle \tag{27}$$

$$[R \;, S] \cdot (X + Y) = [R \cdot X \;, S \cdot Y] \tag{28}$$

$+$ meets $\times$

From both (22) and (24) we easily infer the **exchange law**,

$$[\langle R, S\rangle \ , \langle T, V\rangle] \quad = \quad \langle [R \ , T], [S \ , V]\rangle \tag{29}$$

holding for all relations as in diagram

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "R"'] \arrow[drr, "S"', near end] & A + B & B \arrow[l, "i_2"'] \arrow[d, "V"] \arrow[dll, "T", near start] \\
C & C \times D \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & D
\end{tikzcd}$$

Inductive relations

Example — inductive definition of $\geq$ over the natural numbers: for all $y, x \in \mathbb{N}_0$, define $\mathbb{N}_0 \xleftarrow{\geq} \mathbb{N}_0$ as the **least** relation satisfying

$$\begin{array}{l} y \geq 0 \\ y \geq x \ \Rightarrow \ (y+1) \geq (x+1) \end{array}$$

Thanks to (13), these clauses PF-transform to

$$\begin{array}{rcl} \top & \subseteq & \geq \cdot \underline{0} \\ \geq & \subseteq & suc^{\circ} \cdot \geq \cdot suc \end{array}$$

where $\underline{0}$ denotes the everywhere 0 constant function.

Least prefix points

We reason:

$$\begin{array}{l}
\left\{\begin{array}{lcl} \top & \subseteq & \geq \cdot \underline{0} \\ \geq & \subseteq & suc^\circ \cdot \geq \cdot suc \end{array}\right. \\
\Leftrightarrow \qquad \{ \text{ al-gabr (18) ; coproducts } \} \\
[\top\ , suc \cdot \geq] \ \subseteq\ \geq \cdot [\underline{0}\ , suc] \\
\Leftrightarrow \qquad \{ \text{ “al-gabr” (19) } \} \\
[\top\ , suc \cdot \geq] \cdot [\underline{0}\ , suc]^\circ \ \subseteq\ \geq \\
\Leftrightarrow \qquad \{ \text{ absorption property (28) } \} \\
[\top\ , suc] \cdot (id + \geq) \cdot [\underline{0}\ , suc]^\circ \ \subseteq\ \geq
\end{array}$$

In summary: $\geq$ is the least **prefix** point of monotonic function

$$f\ X \ \triangleq\ [\top\ , suc] \cdot (id + X) \cdot [\underline{0}\ , suc]^\circ$$

Diagrams help

Recognizing $[\underline{0}\ , suc] = in$ as initial $(1 + \_)$-algebra with carrier N_0 (Peano isomorphism) we draw

$$\begin{tikzcd}[column sep=huge, row sep=large]
N_0 \arrow[r, bend left, "in^\circ"] \arrow[r, phantom, "\cong"] \arrow[d, "\geq"'] & 1 + N_0 \arrow[l, bend left, "in"] \arrow[d, "id+\geq"] \arrow[dl, phantom, "\supseteq"] \\
N_0 & 1 + N_0 \arrow[l, "{[\top\ , suc]}"]
\end{tikzcd}$$

$$[\top\ , suc] \cdot (id + \geq)\ \subseteq\ \geq \cdot in$$

Since $[\top\ , suc]$ uniquely determines $\geq$ (least prefix points are unique, etc), we resort to the popular notation

$$\geq\ =\ (\!|[\top\ , suc]|\!) \qquad (30)$$

to express this fact. (See summary of general theory in the sequel.)

Introducing the $\kappa\alpha\tau\alpha$ combinator

In general, for F a polynomial functor (relator) and
$\mu\mathsf{F} \xleftarrow{in} \mathsf{F}(\mu\mathsf{F})$ initial:

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[rr, bend left, "in^\circ"] \arrow[rr, phantom, "\cong"] \arrow[d, "(|R|)"'] & & \mathsf{F}(\mu\mathsf{F}) \arrow[ll, bend left, "in"] \arrow[d, "\mathsf{F}(|R|)"] \\
A & & \mathsf{F}\,A \arrow[ll, "R"]
\end{tikzcd}
$$

there is a unique solution to equation $X = R \cdot \mathsf{F}\,X \cdot in^\circ$ characterized by universal property:

$$X = (|R|) \;\Leftrightarrow\; X = R \cdot \mathsf{F}\,X \cdot in^\circ \qquad (31)$$

(Read $(|R|)$ as "$\kappa\alpha\tau\alpha$ R".)

Introducing the $\kappa\alpha\tau\alpha$ combinator

Therefore (cf. Knaster-Tarski) $(\!|R|\!)$ is both **the least** prefix point

$$(\!|R|\!) \subseteq X \quad \Leftarrow \quad R \cdot \mathsf{F}\, X \cdot in^{\circ} \subseteq X \tag{32}$$

and **the greatest** postfix point:

$$X \subseteq (\!|R|\!) \quad \Leftarrow \quad X \subseteq R \cdot \mathsf{F}\, X \cdot in^{\circ} \tag{33}$$

Corollaries include **reflexion**,

$$(\!|in|\!) \quad = \quad id \tag{34}$$

$\kappa\alpha\tau\alpha$-**fusion**,

$$S \cdot (\!|R|\!) \subseteq (\!|X|\!) \quad \Leftarrow \quad S \cdot R \subseteq X \cdot \mathsf{F}\, S \tag{35}$$

monotonicity,

$$(\!|R|\!) \subseteq (\!|X|\!) \quad \Leftarrow \quad R \subseteq X \tag{36}$$

etc.

Why $\kappa\alpha\tau\alpha$s?

- What's the advantage of writing $\geq \; = (\![\top \; , suc]\!)$? Is it just a matter of *style* or *economy* of notation?
- No: think of proving that $\geq$ is **transitive**:

$$\langle \forall \; x, y, z \; :: \; x \geq y \wedge y \geq z \Rightarrow x \geq z \rangle$$

Instead of providing an explicit (inductive) proof, we go *pointfree* and write:

$$\geq \cdot \geq \quad \subseteq \quad \geq$$

which instantiates $\kappa\alpha\tau\alpha$-**fusion** (35), for $R, X := [\top \; , suc]$.

Thank you, $\kappa\alpha\tau\alpha$-fusion

We reason:

$$
\begin{array}{cl}
 & \geq \cdot \geq \;\subseteq\; \geq \\
\Leftrightarrow & \quad \{ \text{ definition (30) } \} \\
 & \geq \cdot (\![\top , suc]\!) \;\subseteq\; (\![\top , suc]\!) \\
\Leftarrow & \quad \{ \; \kappa\alpha\tau\alpha\text{-fusion (35) } \} \\
 & \geq \cdot [\top , suc] \subseteq [\top , suc] \cdot (id + \geq) \\
\Leftrightarrow & \quad \{ \text{ coproducts (28, etc) } \} \\
 & \geq \cdot \top \subseteq \top \;\wedge\; \geq \cdot\, suc \subseteq suc \cdot \geq \\
\Leftrightarrow & \quad \{ \text{ everything is at most } \top \; \} \\
 & \geq \cdot\, suc \subseteq suc \cdot \geq \\
\Leftarrow & \quad \{ \; \geq \cdot\, suc = suc \cdot \geq \text{ (31) } \} \\
 & \textsc{True}
\end{array}
$$

By the way

Direct use of universal property (31) would lead to

$$
\begin{array}{ll}
 & \geq \; = \; (\![\top \; , suc]\!) \\
\Leftrightarrow & \qquad \{ \;\; (31) \;\; \} \\
 & \geq \cdot [\underline{0} \; , suc] = [\top \; , suc] \cdot (id + \geq) \\
\Leftrightarrow & \qquad \{ \;\; \text{expand, go pointwise, simplify} \;\; \} \\
 & \left\{ \begin{array}{l} y \geq 0 \\ y \geq (x+1) \Leftrightarrow y > 0 \wedge (y-1) \geq x \end{array} \right.
\end{array}
$$

So, the above and our starting (co-inductively flavored) definition

$$
\begin{array}{l}
y \geq 0 \\
y \geq x \;\Rightarrow\; (y+1) \geq (x+1)
\end{array}
$$

are *equivalent* (by construction).

$\kappa\alpha\tau\alpha$ meets fork

What about $\kappa\alpha\tau\alpha$s which are forks? We reason:

$$
\begin{array}{ll}
 & (|\langle R, S\rangle|) \subseteq \langle X, Y\rangle \\
\Leftarrow & \{ \text{ least prefix point (32) } \} \\
 & \langle R, S\rangle \cdot \mathsf{F}\langle X, Y\rangle \cdot in^{\circ} \subseteq \langle X, Y\rangle \\
\Leftrightarrow & \{ \text{ ``al-gabr'' rule (22) } \} \\
 & \left\{ \begin{array}{l} \pi_1 \cdot \langle R, S\rangle \cdot \mathsf{F}\langle X, Y\rangle \cdot in^{\circ} \subseteq X \\ \pi_2 \cdot \langle R, S\rangle \cdot \mathsf{F}\langle X, Y\rangle \cdot in^{\circ} \subseteq Y \end{array} \right. \\
\Leftarrow & \{ \; X := \langle R, S\rangle \text{ in (22); monotonicity } \} \\
 & \left\{ \begin{array}{l} R \cdot \mathsf{F}\langle X, Y\rangle \cdot in^{\circ} \subseteq X \\ S \cdot \mathsf{F}\langle X, Y\rangle \cdot in^{\circ} \subseteq Y \end{array} \right.
\end{array}
$$

Handling mutually recursive relations

- Rule

$$(\!|\langle R, S\rangle|\!) \subseteq \langle X, Y\rangle \quad \Leftarrow \quad \left\{ \begin{array}{l} R \cdot \mathsf{F}\langle X, Y\rangle \cdot in^\circ \subseteq X \\ S \cdot \mathsf{F}\langle X, Y\rangle \cdot in^\circ \subseteq Y \end{array} \right. \tag{37}$$

  tells us how to combine two mutually recursive relations into a single one.

- In the case of functions (20) we get equivalence

$$\left\{ \begin{array}{l} x \cdot in = r \cdot \mathsf{F}\langle x, y\rangle \\ y \cdot in = s \cdot \mathsf{F}\langle x, y\rangle \end{array} \right. \quad \Leftrightarrow \quad \langle x, y\rangle = (\!|\langle r, s\rangle|\!) \tag{38}$$

  known as "Fokkinga's mutual recursion theorem" [2].

- Both (37,38) generalize to $n > 2$ mutually recursive relations (functions) and can be used for program optimization.

Handling mutually recursive relations

- Notice that in° plays no special role in the calculation of (37); so it can be replaced by arbitrary (suitably typed) D.
- This generalizes rule (37) to **divide-and-conquer** algorithms described by recursive relations which are fixpoints of $f\ X \triangleq\ R \cdot (\mathsf{F}\,X) \cdot D$, where R describes the **conquer** step and D the **divide** step.
  (Btw, these are known as *hylomorphisms* [2].)
- For economy of presentation, the example which follows is a direct application of the special case where all relations are functions (38).

Example — exponential function

Taylor series:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{39}$$

Computing finite approximation (n terms)

$$e^x\ n = \sum_{i=0}^{n} \frac{x^i}{i!} \tag{40}$$

takes quadratic time. Wishing to calculate a linear-time algorithm from this mathematical definition, we first head for an inductive definition:

$$\begin{aligned} e^x\ 0 &= 1 \\ e^x\ (n+1) &= \underbrace{\frac{x^{n+1}}{(n+1)!}}_{h_x\, n} + \underbrace{\sum_{i=0}^{n} \frac{x^i}{i!}}_{e^x\ n} \end{aligned}$$

Example — exponential function

We thus get primitive recursive definition

$$\begin{aligned} e^x\ 0 &= 1 \\ e^x\ (n+1) &= h_x n + e^x\ n \end{aligned}$$

where $h_x n$ unfolds to $\frac{x^{n+1}}{(n+1)!} = \frac{x}{n+1}\frac{x^n}{n!}$. Therefore:

$$\begin{aligned} h_x 0 &= x \\ h_x(n+1) &= \frac{x}{n+2}(h_x n) \end{aligned}$$

Introducing $s2\ n = n+2$, we derive:

$$\begin{aligned} & s2\ 0 = 2 \\ & s2(n+1) = 1 + s2\ n \end{aligned}$$

Example — exponential function

We can thus put e^x, $s2$ and h_x together in a system of three mutually recursive functions e^x, $s2_x$ and h_x over the naturals, which PF-transform to

$$
\begin{aligned}
e^x \cdot in &= \underbrace{[\underline{1}\ , (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle]}_{r} \cdot \mathsf{F}\langle e^x, \langle s2_x, h_x \rangle\rangle \\
s2_x \cdot in &= \underbrace{[\underline{2}\ , suc \cdot \pi_1 \cdot \pi_2]}_{s} \cdot \mathsf{F}\langle e^x, \langle s2_x, h_x \rangle\rangle \\
h_x \cdot in &= \underbrace{[\underline{x}\ , (*) \cdot ((x/) \times id) \cdot \pi_2]}_{t} \cdot \mathsf{F}\langle e^x, \langle s2_x, h_x \rangle\rangle
\end{aligned}
$$

respectively, for

$$
\begin{aligned}
in &= [\underline{0}\ , suc] \\
\mathsf{F}\, X &= id + X
\end{aligned}
$$

Example — exponential function

From this system we obtain, thanks to the mutual recursion law (38)

$$\begin{array}{rcl} aux_x & \triangleq & \langle e^x, \langle s2_x, h_x \rangle\rangle \\ & = & \{\ (38)\ \} \\ & & (\!|\langle r, \langle s, t\rangle\rangle|\!) \end{array}$$

for

$$\begin{array}{rcl} r & = & [\underline{1}\ , (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle] \\ s & = & [\underline{2}\ , suc \cdot \pi_1 \cdot \pi_2] \\ t & = & [\underline{x}\ , \underbrace{(*) \cdot ((x/) \times id) \cdot \pi_2}_{u}] \end{array}$$

Example — exponential function

Next we apply the exchange law (29) to $\langle r, \langle s, t \rangle \rangle$ (twice):

$$\langle r, \langle s, t \rangle \rangle \quad = \quad [\langle \underline{1}, \langle \underline{2}, \underline{x} \rangle \rangle \ , \langle (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle, \langle suc \cdot \pi_1 \cdot \pi_2, u \rangle \rangle]$$

Thanks to universal properties (31) and (22) [2] we obtain

$$\begin{aligned} aux_x \cdot \underline{0} &= \langle \underline{1}, \langle \underline{2}, \underline{x} \rangle \rangle \\ aux_x \cdot suc &= \langle (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle, \langle suc \cdot \pi_1 \cdot \pi_2, u \rangle \rangle \cdot aux_x \\ e^x &= \pi_1 \cdot aux_x \end{aligned}$$

that is, we have calculated linear implementation

[2] For functions.

Example — exponential function

```
exp x n = let (e,b,c) = aux x n
          in e where
             aux x 0 = (1,2,x)
             aux x (i+1) = let (e,s,h) = aux x i
                           in  (e+h,s+1,(x/s)*h)
```

which can be identified as the denotational semantics of a while loop, encoded below in the C programming language:

```
float exp(float x, int n)
{
  float e=1; int s=2; float h=x; int i;
  for (i=0;i<n+1;i++) {e=e+h;h=(x/s)*h;s++;}
  return e;
};
```

Summing up

- Algebra of Programming (**AoP**): calculating ("correct by construction") programs from specifications
- Pointfree notation: Tarski's *set theory without variables* [7]
- Kleene algebra of (typed) relations: arrows (not points) provide further structure while ensuring type checking
- *Ut faciant opus signa*:

> [Symbolisms] *"have invariably been introduced to make things easy. [...] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [...] Civilisation advances by extending the number of important operations which can be performed without thinking about them."*
>
> *(Alfred Whitehead, 1911)*

However

Despite textbooks such as [2], **Algebra of Programming** is still land of nobody. Why?

- Software theorists: too busy with their pre-scientific theories (if any)
- Algebraists: not sufficiently aware of program construction as a mathematical discipline
- Both: the required background (categories, allegories, etc) is most often found missing from undergrad curricula.

Selected topic of interest

- Pointfree notations are emerging elsewhere in the context of eg. digital signal processing (SPIRAL project, CMU [6]) which abstract linear signal transforms in terms of (index-free) matrix operators.
- Kleene algebras scale up to the corresponding **matrix Kleene algebras** [1]
- Parallel with relational algebra is obvious.
- Following a similar path, we want to investigate the "matrices as arrows" approach purported by **categories of matrices** (PhD project).
- We believe a better (typed!) calculus of (Kleene) matrix algebras will emerge which will improve reasoning about linear transforms in DSP, divide-and-conquer algorithms, etc.

References

R.C. Backhouse.
Mathematics of Program Construction.
Univ. of Nottingham, 2004.
Draft of book in preparation. 608 pages.

R. Bird and O. de Moor.
Algebra of Programming.
Series in Computer Science. Prentice-Hall International, 1997.

J. Desharnais and G. Struth.
Domain axioms for a family of near-semirings.
In *AMAST*, pages 330–345. 2008.

P.J. Freyd and A. Scedrov.
Categories, Allegories, volume 39 of *Mathematical Library*.
North-Holland, 1990.

Dexter Kozen.
A completeness theorem for Kleene algebras and the algebra of regular events., 1994.
Information and Computation, 110(2):366–390.

Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel.
Formal datapath representation and manipulation for implementing DSP transforms.
In *DAC*, pages 385–390, 2008.

A. Tarski and S. Givant.
A Formalization of Set Theory without Variables.
American Mathematical Society, 1987.
AMS Colloquium Publications, volume 41, Providence, Rhode Island.